SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 101–116 (1996)

Practice

Effective Size: An Example of Use from Legacy Systems

NED CHAPIN
InfoSci Inc., Box 7117, Menlo Park, CA 94026-7117, U.S.A.

TONY S. LAU
Packaged Business Solutions, Inc., 2151 Salvio Street #310, Concord, CA 94520, U.S.A.

SUMMARY

As part of a quality assurance effort directed at legacy systems, we examined a high-cost hard-to-maintain COBOL program. Although written originally in 1968, it had been restructured in 1985 and has been well maintained since. It has the reputation of being 100 per cent well-structured in the software engineering sense, of being very reliable in execution in overnight batch, but of being expensive and difficult to maintain. Our examination covered cyclomatic complexity, iteration, language use, IF-nesting depth, program structure and hierarchical depth. Our most significant finding was the effective size of this nearly 600 module program. Instead of having an effective size (exampled in the paper) of nearly 1·0, its effective size exceeded 13000! This extreme effective size made the program much harder to understand when in maintenance than the program's source-list size and other characteristics would suggest. Our findings included identifying the cause of the extreme effective size. From our experience with this program, we learned seven lessons applicable to improving the maintenance of legacy systems.

KEY WORDS: program comprehension; software size; program structure; conditionals; software quality; software maintenance; structured programming; structured design; software metrics

1. ASSIGNMENT

As part of an organization-wide cost reduction effort, Information Systems management was asked to identify legacy systems having high maintenance costs. The organization, a public utility in the American mid-west, recognizes the importance of its legacy systems to its operating capability and financial health, but questions why some systems have high maintenance costs per maintenance request, when others do not. In the organization, software maintenance teams do the maintenance work.

We worked with a Quality Assurance Team ('QAT') to examine a computer program that had a history of high maintenance costs per maintenance project done. The QAT for this study project reported to the Chief Information Officer's ('CIO's') Administrative Assistant, our 'boss'. In making the project assignment, our boss said she expected the problem was one of program comprehension (see Appendix). To that end, she asked for answers to two questions about this program ('the Program'):

CCC 1040–550X/96/020101–16

Received January 1994
Revised November 1995

1. Why is maintaining the Program so costly? and
2. What could identify programs, during both development and maintenance, with characteristics like those of the Program, to warn of possible high maintenance cost?

The Program is a legacy program in COBOL running overnight six nights a week in batch. The Program's role is to consistency-check and convert as needed data which have been accumulated in a daily transactions file, sometimes using data from tables and other files. The Program is part of a suite of inventory management programs, some interactive and some batch. The Software Development Team ('SDT') wrote the original version of the Program in 1968, and then rewrote it using software engineering techniques including structured design and structured programming methodologies in 1985 (Marciniak, 1994). The 1985 version is the version currently running, and was the subject of our assignment. The Program has not suffered an abnormal termination in execution for years, and rarely needs enhancement. The Program has the reputation of being very hard to understand, and management has identified it as a 'high-cost to maintain' program based upon the cost per instance, not the frequency, of the maintenance work done. In this report, we use software engineering terminology (such as 'module') rather than COBOL jargon (such as 'paragraph') wherever possible.

From a development perspective, the SDT has expressed pride and satisfaction about the Program. The SDT points out that when it turned over the Program to its designated Software Maintenance Team ('SMT') after the 1985 redevelopment:

- The Program was 100 per cent structured (Sommerville, 1992), a figure supported by using the same software tool that the QAT itself uses in checking software quality.
- The Program had a maximum IF nesting depth of three, and only two instances of that.
- The Program was free of GO-TOs, except *within* eleven modules where GO-TOs implement iteration not otherwise possible in COBOL, and except for two modules where GO-TOs implement non-structured links between modules as approved in the organization's SOP (Standard Operating Practices) Manual.
- The Program had no dead (unreachable) code.
- The Program had explicit invocation in every instance of all modules (none invoked implicitly, as for example, by fall-through).
- The Program had no intersecting iterative loops.
- The Program had no code switches (instructions altering other instructions).
- The Program used consistent mnemonic data names.
- The Program had an average cyclomatic number per module of only 4·5.
- The Program was less than half the maximum program-size recommendation of 1200 modules specified in the organization's SOP.
- The Program used numbered modules listed in source-code order to make finding them easy in the source code.
- The Program has a history of very reliable performance in execution.

From a maintenance perspective, the SMT has expressed concern and dissatisfaction about the Program. The SMT points out that:

- The Program has had a high cost per maintenance request, in contrast to the

usual relatively low maintenance costs experienced by the organization for well-structured programs.

- The SMT has preserved all of the SDT-provided qualities (see list above), except for the number of modules.
- The Program now consists of 598 modules, 18 of which since 1985 have been inserted (added to the Program) or completely rewritten by the SMT.
- The Program has more than eight times the SOP's maximum recommended number of conditional (IF) imperative statements for its size of 598 modules.
- Modules in the Program are on average smaller in size, in terms of the number of imperative statements, than the SOP's maximum acceptable size.
- SMT members complain that most modules specify small functions to be performed giving a 'fragmented' impression to the Program.
- The Program, at the SDT's own count of 36, has twice the hierarchical depth recommended as the maximum by the SOP for a 600-module program.
- The Program has 3·8 per cent of its nearly 600 modules with cyclomatic complexity numbers (McCabe, 1976) greater than the 12·0 recommended as the maximum by the SOP.

2. FINDINGS

We examined the Program in several dimensions. We checked with the Computer Operations Team. That team classifies the Program as one of the most reliable they run overnight. It has not failed for years to terminate normally. However, the team finds it impossible to predict the running time with reasonable accuracy, and the Program has to be run before some of the report-producing programs can run. The net finding was that the Program runs very reliably while self-adapting to varied input values, more than to varied input volume.

We checked the SDT's claims. In spite of maintenance work done over the years, the SDT's claims appear to still be intact, except for two points on which the SDT itself was involved. One is the presence of two CALL statements which in a non-structured manner terminate program execution. The SDT put both of these into the Program during the 1985 redevelopment. Second is the use of two GO-TOs to provide non-structured links between modules. The SDT also put both of these into the Program during the redevelopment. The effect of these two GO-TOs, because of their placement, appears to be minor on the structure, although their presence technically converts the hierarchy of modules into a network of modules, increasing undesirably both fan-in and coupling (Myers, 1978). The preliminary net finding was that the Program still substantially satisfies most of the SDT's claims, and has not suffered deterioration in structuredness during maintenance.

We checked the SMT's claims. We examined the 18 modules that the SMT had added or completely rewritten. These modules have raised the current value of the Program's average cyclomatic number from 4·5 to the current value of 4·6. This is because the average cyclomatic complexity for those 18 modules is 8·3. Also, those 18 modules include the two modules with cyclomatic numbers of 33 (the maximum in the Program), and one with 20. Thus, the SMT has contributed 16·7 per cent of the modules with cyclomatic numbers greater than the maximum of 12·0 recommended by the SOP, compared to the SDT's contribution of 2·1 per cent. Put another way, the 3·1 per cent

of the Program's modules added or rewritten by the SMT provide 5·7 per cent of the Program's total cyclomatic complexity. Our finding was that the Program's cyclomatic complexity has been rising as a result of maintenance, but still is less than the typical roughly 8·0 for legacy software.

We analysed the language use. Table 1 shows the basics about the Program. The outstanding features are the skimpy use of data movement and arithmetic statements, and the heavy use of decision and control statements. Within the decision and control group, we could find only 30 iteration control statements. We found four of the modules implementing iteration with fourteen of the GO-TOs in a manner violating the proper

Table 1. Description of the program

Source code allocation	
Identification division — 19 lines	
Environment division — 71 lines	
Data (declarations) division — 2687 lines	
Procedure (imperatives) division — 9049 lines	
Total amount of source code — 11826 lines	
Imperative allocation	
Input and output commands (5%)	
241	Display
25	Close
23	Open
14	Write
8	Read
3	Rewrite
2	Accept
Data movement commands (22%)	
1357	Move
8	Sort
3	Unstring
Arithmetic commands (7%)	
233	Add
148	Divide
4	Subtract
3	Compute
Logic Commands (0%)	
8	Inspect
Decision and control commands (66%)	
2169	IF
1847	Perform (non-iterative)
24	GO-TO (iterative)
4	Perform until (iterative)
4	GO-TO (non-iterative)
3	CALL
1	Return
1	GOBACK (termination)
6133	Total number of commands

structure expected in software engineering (von Mayrhauser, 1990). Figure 1 gives an unstructured example from the Program.

In analysing language use, we found overall 2 169 conditional statements (IFs). It is rare to have a high percentage of conditional statements and a low percentage of iteration control statements present with a relatively deep hierarchy in a non-trivial program. Our net finding on language use was that the Program does highly conditional data editing with little explicit iteration and without keeping proper (in the software engineering sense) structure in the control of the iteration.

We analysed the relationships among the conditional imperative statements (mostly IFs) in the Program, since the SDT's claim is ambiguous. The claim could apply to the modules either individually or collectively. From software engineering, we know that all well-structured programs must have all the binary-choice equivalents of the present conditional imperatives in one of the three structures: sequence, selection or iteration. We

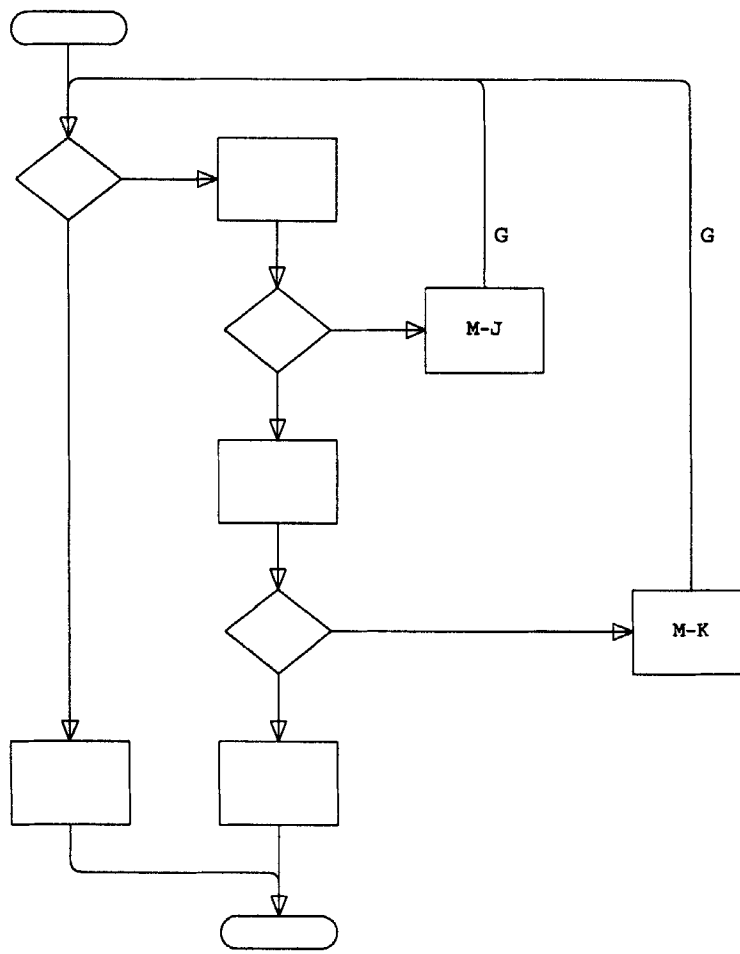


Figure 1. This example of a module from the Program uses GO-TOs (indicated by **G***) for invoking the unstructured iteration of modules M–J and M–K. This module uses only two GO-TOs; the other unstructured iteration-directing modules use from three to five GO-TOs in the Program*

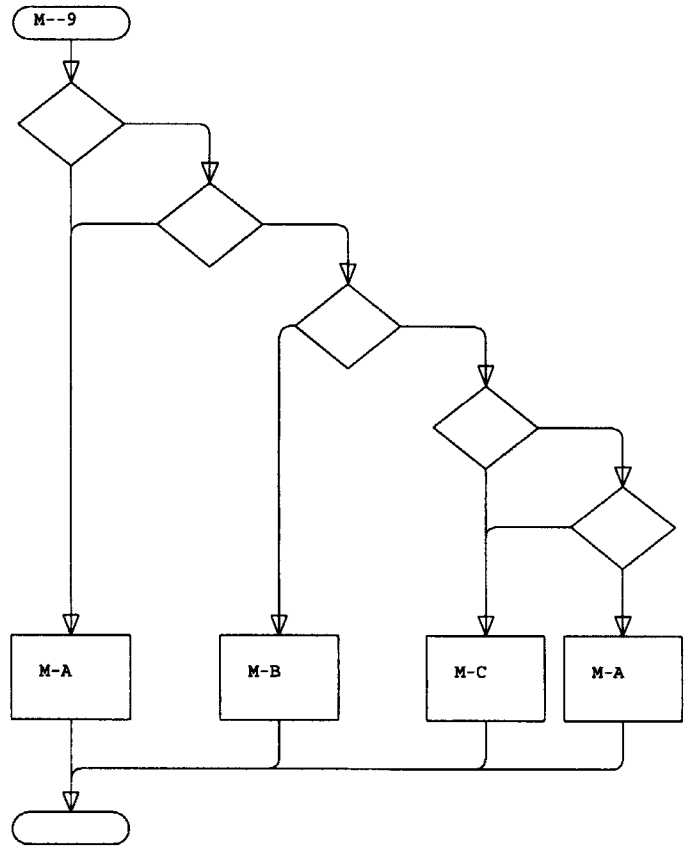
also noted that nine modules had cyclomatic number complexities of 20 or more. By their nature, well-structured implementations of structured designs are deeply nested. Hence, it is highly improbable that a 600-module program would have a maximum IF-nesting-depth of three. To check the claim, we counted the actual IF nesting depth of all modules, and of the structure of modules comprising the longest path in the Program. We found that the actual maximum IF-nesting depth in any one module is 14, and in the Program as a whole is at least 155. Our net finding was that the Program conditional nesting depth maximum is actually more than 150 which is at least 50 times greater than the level of three claimed by the SDT for the Program. Further, the module maximum nesting depth is nearly five times that ambiguously claimed level of three ('ambiguously' since the SDT also speaks of 'instances'). Both of these findings ignore the effects of the very numerous 'AND' and 'OR' compounding in the conditional statements.

We examined how the SDT-claimed maximum IF-nesting of three could be supported. We noted that the SDT recognized a nesting depth increase when a conditional appeared immediately following another conditional without an intervening 'ELSE'. Otherwise, the 'pretty printing' of the source code kept the conditional statements lined up vertically, giving an impression of no nesting. Figure 2 give an example from the Program of one of the modules claimed to have an IF-nesting depth of one, again ignoring the presence of 'AND' and 'OR' compounding. Our net finding was that the Program is expressed in a manner that obscures much of the presence and consequences of nested conditional statements. This requires SMT personnel (the usual readers of the source code) to add a mental translation process to interpret what appears to be non-nested conditional statements, as executing actually as nested conditional statements, sometimes very deeply nested in the Program.

We analysed the structure of the Program. Looking at the structure of modules as a hierarchy, we found the actual maximum depth of the structure to be 41 levels. This is five more than the 36 the SDT had claimed, and was for a part of the structure unchanged by the SMT. From software engineering and the SDT's claims that the Program was well-structured and had no dead code (which we confirmed), we could have expected to find the Program to have an effective size of 1·00 (see Section 3). However, we actually expected to find the Program would have an effective size of about 1·05—that is, execute as a structure of about 630 modules. We expected this increase above 1·00 because of the presence of the non-structured CALLS and GO-TOs (two of each). Instead we were astounded to find that the Program has an effective size of 13 624·15! This means the SMT has to understand the Program as it executes as if it were a program of 8 147 242 modules—*more than eight million* modules! This was totally unexpected. Disbelieving, we checked but found the count to be correct. Our net finding was that the Program has the characteristics of a program more than thirteen thousand times larger than it appears to be from the program listing.

3. EFFECTIVE SIZE

As known from software engineering, a program may execute at a maximum as a number of modules different from a count of the actual number of distinct modules. At the least (and suppressing selection and iteration, and assuming no 'dead code') the maximum number of modules executed is the same as (is divided by) the actual number of distinct modules, for an effective size ratio of 1·00. Effective size has not been identified heretofore




```
M--9.
    IF D1 NOT EQUAL D2
        PERFORM M-A
    ELSE
    IF D3 NOT EQUAL D4
        PERFORM M-A
    ELSE
    IF D1 NOT EQUAL D5
        PERFORM M-B
    ELSE
    IF D6 EQUAL D7
        PERFORM M-C
    ELSE
    IF D8 EQUAL D9
        PERFORM M-C
    ELSE
        PERFORM M-A.
```

Figure 2. This example shows both the flowchart and the COBOL source code for a module from the Program. This module was rated by the SDT as having an IF-nesting depth of one, but actually has a depth of four. The SMT has not modified this module; it is as the SDT rewrote it in 1985. The data and module names have been simplified for this figure (M–1–1–5–3–9)

in the literature as a major influence on program comprehension. An example can help clarify the concept of effective size. Figure 3 gives an example in structure chart form for a program of 10 distinct source-coded modules in a hierarchy of three levels.

Let us examine Figure 3 in detail.

- Each of the leaf modules (the ones at the third (bottom) level with the three-digit names) invokes in execution no other modules (has no children). Hence, each leaf module has a count of the modules executed equal to one more than the sum of the count of the modules executed, zero, by its zero children. Each module executes only itself; hence the 'one more than' addition to the count. The count of the modules executed by each module is noted on the structure chart just to the left of each module.
- At the middle level, module 1.3 (the third child of the root module) invokes its one child, module 1.3.1, when module 1.3 is invoked. Hence, the structure of modules headed by module 1.3 has a count of the modules executed of two, one more than the count of the modules executed by its one child.
- Module 1.1 (the first child of the root module) invokes its two children, modules 1.1.1 and 1.1.2, when it is invoked. Hence, the structure of modules headed by module 1.1 has a count of the modules executed of three, one more than the sum of the count of the modules executed, one each, of its two children.
- Module 1.2 (the second child of the root module) invokes its three children, modules 1.2.1, 1.2.2 and 1.2.3, when it is invoked (and suppressing both selection and iteration). Hence, the structure of modules headed by module 1.2 has a count of the modules executed of four, one more than the sum of the count of the modules executed, one each, of its three children.
- At the first or top (root) level, module 1 invokes its three children, modules 1.1,

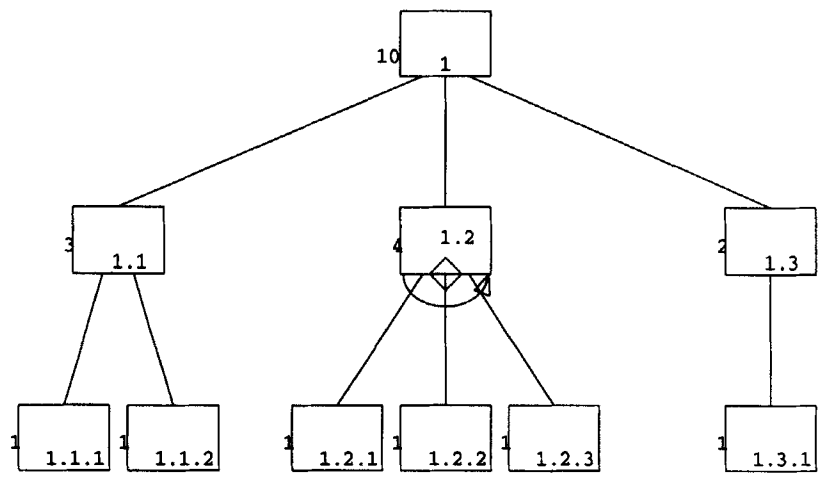


Figure 3. The program diagrammed in this structure chart consists of 10 modules. Since it also executes as a program of 10 modules, it has an effective size of 1·00. Note that the effective size counting process suppresses the iteration and the selection (both are shown under module 1.2). Module by module, the count of modules executed is shown in this figure just to the left of each module for that module and all the modules in the subtree below it

1.2 and 1.3, when it is invoked. Hence the structure of modules headed by module 1 has a count of the modules executed of 10, one more than the sum of the count of the modules executed, three, four and two respectively, of its three children.

The 10-module program in Figure 3 executes, suppressing selection and iteration, as a structure of 10 modules, and hence has an effective size of 1·00 (10 divided by 10). The arrowed arc (loop) under module 1.2 indicates iteration of the attached structure of child modules 1.2.1, 1.2.2 and 1.2.3 (Myers, 1975). 'Suppressing iteration' means *not* counting *each* possible time that executing an iteration could result in invoking a module, but instead counting iterated modules *once*. The diamond under module 1.2 indicates conditional execution (selection) of child module 1.2.2. 'Suppressing selection' means ignoring the possible effect of selection, and counting the modules as though no selection were present. Thus, in Figure 3, the count of the modules executed remains the same at 10 (because of the suppressing selection and iteration convention), regardless how many times the iteration might occur in an actual execution and regardless of how many modules might be skipped (not selected) in an actual execution.

Since 'straight-line code' executes more rapidly than does iterated code, some programmers sometimes reduce 'overheads' by writing out explicitly enough code to provide for the maximum possible number of iterations. Each function, whether or not repeated, gets its own distinct module. For example, if the maximum number of iterations were three, then Figure 4 shows a hierarchy of three levels for exactly the same job as does Figure 3. Each connecting line indicates an invocation link. Iteration is suppressed, since none now appears (note the absence of the arc indicating iteration). Now, the program hierarchy in Figure 4 executes as a structure of 16 modules, because it has a larger number of modules present in the source code (including module versions A, B and C) to do the same total work done in Figure 3. Note that this explicit writing out as a substitute for iteration, increases the count of the modules executed. By itself, however, this explicit writing out of modules still makes the count of the number of modules executing in the structure (suppressing iteration) be equal to the number of distinct modules in the source code. In Figure 4, both counts are of 16 modules, for the same work as 10 modules specified in Figure 3, and for the same effective size, 1·00.

If not all of the functions need to be invoked on every iteration, the usual practice in software engineering is to put the needed control or decision making in the iteration control module (Myers, 1975). This is module 1.2 in Figure 3. The presence of such decision making and conditional execution is often indicated by the use of small diamonds on the parent end of the connecting lines, as shown between modules 1.2 and 1.2.2 in Figure 3 (Myers, 1975). Where well-structuredness is not being observed, an alternative is to convert the hierarchy into a network, by reallocating functions among modules and adding additional invocation links. Figure 5 shows a example reworking of the same 10 distinct modules as in Figure 3, with the additional links shown as curving (not straight) lines. Note that no iteration appears because the decision making and added invocation links substitute for the iteration. As before, the number to the left of each module in Figure 5 indicates the count of the modules executed by the substructure of modules headed by that module. In Figure 5, a structure of 10 distinct modules executes, suppressing selection and iteration, as a structure of 20 modules. That is, it has an effective size of 2·00. The network structure which results hinders breaking or slicing the program up into loosely coupled chunks or clusters of modules for easy understanding. Also, the network

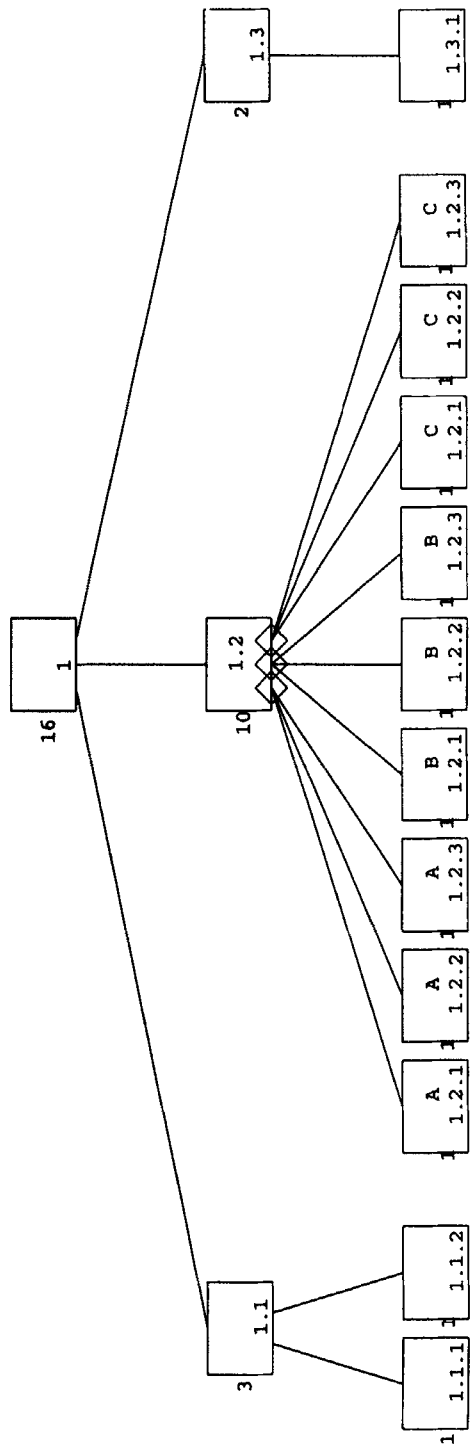


Figure 4. This structure chart converts the program of Figure 3 into 'straight-line' form, assuming that the maximum number of iterations possible is three, shown as A, B and C in the figure. As before, the count of the modules executed is shown just to the left of each module

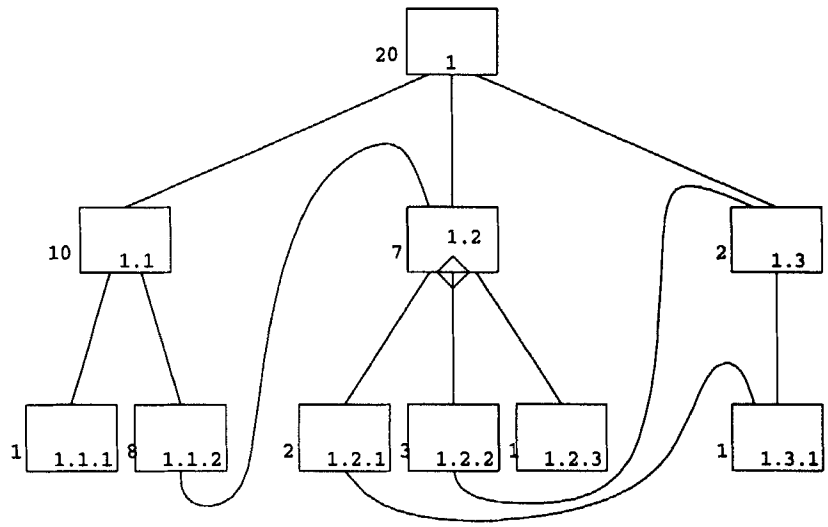


Figure 5. This structure chart shows a program of 10 modules (the same modules as in Figure 3) structured to have a count of modules executed of 20. As before, the number shown to the left of each module is the count of the modules executed by the substructure headed by that module. The effective size is now 2·00 (20 divided by 10)

structure which results is never well-structured in the software engineering sense, and typically provides examples of the classic forms of non-structured software.

4. CONCLUSION

The pattern diagrammed in Figure 5 is the archetype for the patterns observed in the Program, a portion of which is diagrammed in Figure 6. Hence, our main conclusion or net finding is that the Program is grossly lacking in well-structuredness in its structure of modules. In spite of the SDT's claim, the Program is actually much closer to zero per cent well-structured than it is to 100 per cent well-structured, as the extremely large effective size testifies.

We did not have time to examine other attributes and properties of the Program beyond those reported here. So we cannot report on such aspects as compound conditional use, hard-coded constants, functionality, iteration control, efficiency in hardware use, and data utilization. Nor did we have the time or authorization to delve in detail into the history of the Program to find out who did what and when, and who approved what and when. We were able to find the file of SOPs, however, and examine it.

Our findings provide answers to the boss' two questions posed in our QAT job assignment about this legacy program (the Program). Those questions and their answers are:

- **Question 1**. Why is maintaining the Program so costly? **Answer**: The Program is *not* a well-structured program, sincere claims to the contrary not withstanding. The non-structuredness in this restructured legacy program partly takes the form of a huge effective size for the Program. Just being labelled 'well-structured' does not make a program well-structured. In fact, since the Program is labelled as being well-

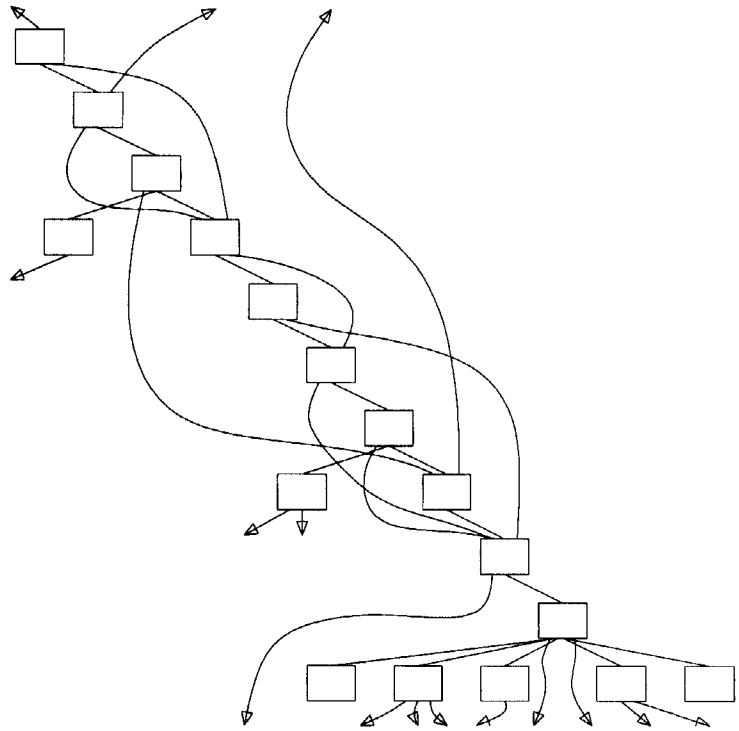


Figure 6. This portion of a structure chart shows the modules in the lower-middle part of the deepest spline of the Program. The chart shows all control (execution) links starting or ending at each of the modules shown. The highest module shown is at level 22 (M–1–1–5–3–9–2–10–4–3–1–1–1–1–3–3–4–1–1–1–1–1–1)

structured, the SMT personnel very probably suffer without realizing it, an additional barrier to achieving program comprehension (Letovski and Soloway, 1986). They probably use an inappropriate mind-set in trying to understand the Program. They expect to find certain specific characteristics and attributes, but strangely what at first looks like them, turns out not to act like or be them.

- **Question 2.** What could identify programs during both development and maintenance, with characteristics like those of the Program, to warn of possible high maintenance cost? **Answer**: Verify that claims of well-structuredness, especially for legacy programs, are founded on fact, both before and after performing either maintenance or redevelopment work. Verify the presence of well-structuredness also at intake of programs into maintenance. Both of these could be QAT functions, and could use quantitative metrics, including effective size, for example.

5. LESSONS LEARNED

In retrospect, those answers were less valuable than some lessons learned. We and the organization learned seven lessons from the job assignment applicable to improving the maintenance of legacy systems:

1. A legacy program may have an effective size much larger than its expected effective

size. In this case, the Program consisting of nearly 600 modules had an expected effective size of a little over 1·0, but had an actual effective size of more than 13 500. The actual effective size can be a useful indicator of the likely difficulty of understanding a program, especially when used with the source-code size of the program. The effective size metric is quantitative, language independent, objective and program size independent, but assumes a software engineering environment for its interpretation. It meets the usual tests for a software metric of complexity (Zuse, 1990). The effective size metric is fairly easy to calculate by computer. When all other factors are equal, larger programs are more difficult for programmers and analysts to understand, and add more to the cost of doing software maintenance (see Appendix for a summary on program comprehension).

2. A legacy program that appears to have conditionals in simple sequences, may actually execute those conditionals in a nested pattern, sometimes even deeply nested. In this case, the Program rarely appeared to have an IF nesting greater than two. Yet, in execution, nearly all IFs executed in a deeply nested pattern, sometimes exceeding 150 levels of conditional nesting depth. Understanding the relationships among conditionals typically is one of the most time consuming parts of doing software maintenance. Not showing the conditional context of the conditionals hinders the process of building understanding by representing the conditionals as something they are not.
3. Using substitutes for iteration in a legacy program can complicate the program, making it more difficult to understand (Iselin, 1988). In this case, the Program reduced explicit iteration in two ways. One was by converting a hierarchy of modules into a network of modules while reallocating functions among the modules. Another was by keeping iteration out of sight within the modules. As software engineering points out, the abstraction process which recognizes iteration, gives economy and elegance to the form and expression of the program (McGowan and Kelley, 1975; Sommerville, 1992). Programmers and analysts have been trained to look for, expect, and use iteration explicitly. Obscuring iteration in a program impedes understanding and thus raises the cost of maintenance.
4. Program documentation may convey facts or interpretations, but may leave for the reader to ascertain which is which. A structure chart, prepared to an accepted set of conventions, typically provides facts, for example. A narrative description which states a degree of structuredness for a program, typically provides interpretation, for example. In this case, the Program's documentation provided an interpretation of structuredness, an interpretation unsupported by the facts. To help maintenance personnel do their work on legacy systems, the documentation of facts provides a better basis for maintenance action than does a documentation of interpretations.
5. The maintenance intake (acceptance) procedure should be more than an automatic rubber stamp, even for legacy systems. To identify high-cost software, the turnover process for the acceptance into maintenance status of software from development (in-house or vendor), has to look at the facts. In this case, the QAT had apparently based its acceptance of the Program upon the SDT's assurances that the Program was 100 per cent structured. Actually, the Program failed to meet the SOP for acceptance (both as it existed in 1985, and currently), and should have been rejected as not ready for acceptance into maintenance.
6. Software tools do not always do the job expected of them in working on legacy

systems. In this case, the SDT used the output from a software tool (one also used by the QAT), to help assess the structuredness of the Program. The tool either failed to detect the Program's non-structuredness, or both the QAT and the SDT failed to interpret the tool's output correctly.

7. The obvious question to ask is not always the best question to ask in maintaining legacy systems. In this case, the boss recognizes she should have asked 'How does the SMT manage to maintain at such a low cost what is effectively an eight million module program?' The boss says she will leave that matter for a future project, and in the meantime hope that the number of maintenance requests continue to be relatively rare for the Program.

Acknowledgements

We thank the boss and the organization for their support, and the anonymous reviewers for their counsel.

APPENDIX: PROGRAM COMPREHENSION

Aspects

People working with legacy systems have to contend with program comprehension difficulties at least as much as people working with other systems. Since understanding the system is the most skilled-labour-intensive part of software maintenance, difficult program comprehension adds to the cost software maintenance. Program comprehension has five main aspects (Corbi, 1989; Curtis *et al.*, 1989): Domain, Software, Representation, Characteristics and Psychology.

Domain

The domain of the legacy system is the business, industrial, government, engineering, or science area where people use the system. Examples are accounts receivable, oil refining, machine tool control, biomedical monitoring, driving license registration and image enhancement. In doing software maintenance on legacy systems, a thorough knowledge of the domain is usually very helpful, but even people who are experts in any specific domain, have only incomplete domain knowledge. Inadequate, inaccurate, and out-of-date domain knowledge hamper the maintenance of legacy systems.

Software

Some people concerned with program comprehension regard software narrowly as the program or specified sequence of steps people give a computer to direct its performance as part of a system. Other people regard software more broadly by also including all the other non-hardware components of a system that has a computer as a component.

Representation

For people to communicate domain knowledge, they have to have a way of representing that knowledge. Furthermore, software is intangible, and it is only the representation of

the software which can be communicated between people and between people and computers. Many means of representation can be used for these situations. Service requests, data flow diagrams, source code listings, tax regulations, requirements statements, mathematical formulas, narratives, IDEF diagrams, data layouts and photographs are examples. The poor quality of the documentation for legacy systems is a common complaint.

Characteristics

The domain has characteristics, the software has characteristics, and the means of representation also has characteristics. These act in combination, sometimes to make the complex appear simple, and other times to make the simple appear complex. For example, on legacy software (and ignoring domain matters), some of the common characteristics are source language used (such as FORTRAN, COBOL, etc.), size of the software (measured in various ways, such as the number of lines of source code), detail level in a flow chart, data naming conventions used, and type of annotation in the source code. Some characteristics contribute to easy understanding (comprehension), such as small size, little conditionality, well-known language used in usual ways, readable accurate documentation and small number of variables (input, output and internally within the program).

Psychology

Program comprehension is something people do, and people are very varied. For some, a picture is worth a thousand words; for some others, words tell best; for a few, tables speak compellingly, and for still others, nothing beats mathematics. Biologically, peoples' brains just do not all work the same way. In addition, people learn skills and develop preferences. Since people have different educational and training backgrounds, they have learned more about some domains, some software, some representations, some characteristics, and hence feel more confident and comfortable with them. On top of all that, people have different experiences. Some experiences lead to aversion, others to acceptance and skill building in various domains, using an assortment of software and representations. Even when the domain, software, representation, and characteristics are the same, the people are different, and hence the program comprehension is different.

References

Corbi, T. A. (1989) 'Program understanding: challenge for the 1990's', *IBM Systems Journal*, **28**(2), 294–306.

Curtis, B., Sheppard, S., Kruesi-Bailey, E., Bailey, J. and Boehm-Davis, D. (1989) 'Experimental evaluation of documentation formats,' *Journal of Systems and Software*, **9**(2), 167–207.

Iselin, E. R. (1988) 'Conditional statements, looping constructs, and program comprehension: an experimental study', *International Journal of Man–Machine Studies*, **28**(1), 45–66.

Letovski, S. and Soloway, E. (1986) 'Delocalized plans and program comprehension', *IEEE Software*, **3**(5), 41–49.

Marciniak, J. J. (Ed.) (1994) *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., New York, NY, 1453 pp.

McCabe, T. J. (1976) 'A complexity measure', *IEEE Transaction on Software Engineering*, **SE–1**(3), 312–327.

McGowan, C. L. and Kelly, J. R. (1975) *Top-down Structured Programming Techniques*, Petrocelli Books, New York, NY, 228 pp.

Myers, G. J. (1975) *Reliable Software Through Composite Design*, Petrocelli/Charter Publishers, Inc., New York, NY, 159 pp.
Myers, G. J. (1978) *Composite/Structured Design,* Van Nostrand Reinhold Co., New York, NY, 174 pp.
Sommerville, I. (1992) *Software Engineering*, Addison–Wesley Publishing Co., Inc., Reading, MA, 649 pp.
von Mayrhauser, A. (1990) *Software Engineering*, Academic Press Inc., San Diego, CA, 864 pp.
Zuse, H. (1990) *Software Complexity Measures and Methods*, Walter de Gruyter & Co., New York, NY, 605 pp.

Authors' biographies:

Ned Chapin serves as an Information Systems Consultant. He has a keen interest in how and why some organizations get so much better value from their information systems than do other organizations. Ned draws upon experience in systems analysis and design, programming, database, testing, maintenance, auditing, management, documentation, accounting, and engineering. Ned holds an MBA from the University of Chicago, a Ph.D. from Illinois Institute of Technology, and certificates and licenses.

Tony S. Lau is a programmer/analyst in California. Currently he is pioneering a mortgage–specific integrated EDI system to harmonize trading partners' data transfers in a seamless manner. Tony specializes in cross-platform relational database application design, development and maintenance. He has had extensive experience in complex business management applications. Tony has an MBA in Computer Information Systems, and a B.Sc. in Industrial Engineering.